

```

// Computer Program Listing Appendix Under 37 CFR 1.52(e)
// Fig4a.java
// Copyright (c) 2004. Borland Software Corp. All Rights Reserved.
final void readOnlyLogicalRestart(DataStoreConnection con) {
    // Need dataStore monitor before storeOwner (this) because TxCursor.logicalUndo
    // calls writeCon.dataStore.closeDirectory();
    //
    synchronized(dataStore) {
        synchronized(this) {
            if (!hasWriteBlock(WriteBlock.UNVERIFIED_PRIMARY_MIRROR) && !logicalRestart) {
                DataStore dataStore = con.dataStore;
                TxAnchor anchor = dataStore.logMan.anchor;
                long oldestLsn;
                TxCursor cursor = con.logCursor;
                // undoActives initialized in startReadOnlyTx
                //
                // TxActiveList list = cursor.cloneUndoActives();
                oldestLsn = TxCursor.getOldestUndoActiveLsn(cursor.roUndoActives);
                Diagnostic.check(con.storeOwner != con.storeOwner.writeOwner);
                if (oldestLsn == 0)
                    oldestLsn = roOldestLsn;
                Diagnostic.check(oldestLsn != 0);
                if (oldestLsn > 0) {
                    if (this == writeOwner)
                        dataStore.setConsistent(false);
                    Diagnostic.check(anchor.shutdownLsn <= oldestLsn);
                }
                // Diagnostic.check(anchor.mirrorTypeCode == MirrorTypeCodes.PRIMARY || anchor.mirrorTypeCode ==
                MirrorTypeCodes.READONLY);
                Diagnostic.check(shadowTable == null || this == writeOwner || shadowTable.getLongRowCount() == 0);
                createShadowTable(roLastLsn);
                // try {
                con.storeOwner.dirty |= ListenerFlags.SHADOW_TABLE|ListenerFlags.READONLY;
                logicalRestart = true;
                // The physical and logical undo will not leave valid
                // lsns in the blocks they update, so other read only transactions
                // should not try to share them.
                //
                con.storeOwner.dirty |= ListenerFlags.READONLY_NOSHARE;
                cursor.physUndo(cursor.roUndoActives);
                cursor.logicalUndo(cursor.roUndoActives);
                con.storeOwner.dirty &= ~ListenerFlags.READONLY_NOSHARE;
                roLogicalUndoComplete = true;
                // }
                // finally {
                // con.storeOwner.dirty &= ~(ListenerFlags.SHADOW_TABLE|ListenerFlags.READONLY);
                // }
            }
        }
    }
}

```

```

    }
    final void createShadowTable(long lastLsn) {
        Diagnostic.check(allocMapEntry == null || (allocMapEntry.listenerFlags & ListenerFlags.DIRTY) == 0);
// Diagnostic.check(    datastore.logMan.anchor.mirrorTypeCode == MirrorTypeCodes.READONLY
//                    || datastore.logMan.anchor.mirrorTypeCode == MirrorTypeCodes.PRIMARY);
        if (shadowTable == null/* && datastore.logMan.anchor.mirrorTypeCode == MirrorTypeCodes.READONLY*/) {
            if (this == writeOwner && datastore.logMan.anchor.mirrorTypeCode == MirrorTypeCodes.PRIMARY)
                addWriteBlock(WriteBlock.UNVERIFIED_PRIMARY_MIRROR);
// Diagnostic.check(roLastLsn == lastLsn || roLastLsn == 0 || (this == writeOwner && lastLsn > roLastLsn));
            shadowLastLsn = lastLsn;
            shadowEntries = new Vector();
            freeShadowEntries = new Vector();
            shadowTable = new StorageDataSet();
            Diagnostic.check(!dataStore.mirrorVerified || this != writeOwner);
            shadowTable.setResolvable(false);
            String storeName;
            storeName = "ro";
            storeName += hashCode();
            Column col = new Column();
            col.setColumnName("b");
            col.setDataType(Variant.LONG);
            col.setAutoIncrement(true);
            MatrixData.forceAutoInc(col);
            col.setPersist(true);
            shadowTable.addColumn(col);
            col = new Column();
            col.setColumnName("sb");
            col.setDataType(Variant.INT);
            shadowTable.addColumn(col);
            shadowTable.setStoreName(storeName);
            shadowTable.setStore(dataStore._retrieveTempDataStore());
            shadowTable.open();
            shadowListener = ((TableData)MatrixData.getData(shadowTable)).btree;
            shadowOwner = shadowListener.storeOwner;
            Diagnostic.check(shadowTable.getColumn(0).isAutoIncrement());
        }
    }
}

```

// Fig4b.java

// Copyright (c) 2004. Borland Software Corp. All Rights Reserved.

private final CacheEntry _getReadBlock(int block, StoreOwner readOwner, SaveListener saveListener)

/*-throws DataSetException-*/

```

{
    Cache    cache    = readOwner.cache;
    StoreOwner writeOwner = readOwner.writeOwner;
    CacheEntry entry;
    CacheEntry bestEntry = null;
    CacheEntry writeOwnerEntry = null;
    CacheEntry retEntry = null;
    long    highestLsn = 0;
    long    bestLsn    = 0;

```

```

long    bestDelta    = Long.MAX_VALUE;
boolean bestInRange  = false;
CacheOwner bestOwner = null;
long    lastLsn;
long    entryLsn;
long    delta;
boolean inRange;
int     yieldCount;
boolean writeCopy    = false;
int     tempOff;
long    tempLsn;
boolean replace;
boolean fetched = false;
Diagnostic.check(cache == writeOwner.cache);
Diagnostic.check(readOwner.roLastLsn == roLastLsn);
Diagnostic.check(readOwner.roOldestLsn == roOldestLsn);
// long    lastLogLsn = logMan.getLastLsn();
// Calling logMan.getLastLsn() acquires logmMan monitor, leading to thread
// deadlock when connection being open and other monitors like the conMonitor
// are already held by this thread. Shows in TestMirrorTpc test.
//
long    fuzzyLastLogLsn = logMan.anchor.lastLsn;
boolean useBlock;
// long roHighLsn;
// if (block == 68)
//     Diagnostic.println("stop:");
//     Diagnostic.println("getReadBlock: "+saveListener);
// Diagnostic.println("_getReadBlock: "+block+" "+roOldestLsn+" "+Thread.currentThread().hashCode());
synchronized(cache) {
    entry    = cache.getBlockList(block);
    Diagnostic.check(writeOwner != readOwner && !writeOwner.readOnlyTx);
    while (entry != null) {
        // Note that another readonly connection thread could be waiting for a copy, so don't look at such a
        // block (it will have a 0 LSN!).
        //
        if (entry.block == block && readOnlyCanUse(writeOwner, entry)) {
            if (entry.owner == writeOwner) {
                writeOwnerEntry = entry;
            }
            if (canCopy(writeOwner, entry)) {
                // canCopy should only return true if it is known that entry.buf
                // log id bytes have been completely initialized at least once.
                // It is possible for writeOwner blocks to be in the process
                // of having the log id rewritten. This writing happens from
                // left to right, so the lsn retrieved may not be a correct
                // address, but will be >= to the initial address that entry.buf
                // was initialized to.
                entryLsn = Packer.unpackLong(entry.buf, BlockHeader.LOG_ID);
            }
        }
        else

```

```

    entryLsn = 0;
Diagnostic.check( (entryLsn >> 48) == 0 ? null :
    " entry.block: " + entry.block
    + " entryLsn " + Long.toHexString(entryLsn)
    + " highestLsn " + Long.toHexString(highestLsn)
    + " owner == writeOwner " + (entry.owner == writeOwner)
    + " cacheFlags " + Integer.toHexString(entry.cacheFlags)
    + " listenerFlags " + Integer.toHexString(entry.listenerFlags)
    + " cacheListener " + entry.cacheListener
    + "" + BlockLog.dump(Diagnostic.out, "dump:", entry.buf, 0, entry.buf.length));
if (entryLsn > highestLsn)
    highestLsn = entryLsn;
// If not less than current log file. Important if we bring old page
// up to new page using redo - must know that log is not going to be
// deleted.
//
// One exception is when this is a writeOwner block. If no other blocks
// are found then the block will be fetched ( read from disk). The
// fetch block can have an older Lsn than the writeOwner block in
// the cache. So the writeOwner block must be considered. Note that
// this was discovered when this block was read by StreamVerifier access
// of a block that was modified by a crash recovery. Block on disk
// had an invalid FileSlot since it did not have redo records applied
// from crash recovery. Note that the writeOwner block will always
// have an Lsn >= the fetched block, so it is always the better choice.
//
if ( entry.owner == writeOwner
    || (entryLsn >> 32) >= ((roOldestLsn >> 32))) // -DataStoreConst.READONLYTX_SAVE_LOG_COUNT))
{
    useBlock = true;
}
else
    useBlock = false;
if (useBlock && entryLsn > 0) {
//     if (Diag.CHECK && entryLsn <= 0)
//         Diagnostic.println("entryLsn:");
    if (entryLsn >= roOldestLsn && entryLsn <= roLastLsn)
        inRange = true;
    else
        inRange = false;
    if (entryLsn < roOldestLsn) {
        if (entry.owner == writeOwner || (roOldestLsn - entryLsn) < ((fuzzyLastLogLsn - entryLsn) / 4)) {
            // Undo should typically traverse less log records than redo,
            // so give less weight to a redo.
            //
            tempOff = (int) entryLsn;
            tempLsn = (entryLsn & (BlockHeader.LOG_ID_MASK)) + tempOff / 4;
            delta = roOldestLsn - tempLsn;
        }
        else {

```

```

        // Better to just refetch the block from disk.
        //
//      Diagnostic.println("REFECTH FROM DISK: "+entry.block);
        useBlock = false;
        delta = Long.MAX_VALUE;
    }
}
else
    delta = entryLsn - roOldestLsn;
if (!useBlock)
    replace = false;
else if (bestLsn == 0)
    replace = true;
else if (inRange) {
    if (bestInRange)
        replace = entryLsn > bestLsn;
    else
        replace = true;
}
else {
    if (bestInRange)
        replace = false;
    else
        replace = delta < bestDelta;
}
if (replace) {
// NOTE THAT I THINK THIS IS NO LONGER TRUE DUE TO THE USE OF SHADOW TABLE
// THAT FIRST PERFORMS A LOGICAL UNDO. IN THE OLD WAY, ALL UNDO WAS PHYSICAL
// DIRECTORY WAS DIFFERENT FROM OTHER TABLES WHEN ROWLOCKS WERE NOT SUPPORTED
// IN THAT THEY COULD NOT USE TABLE LOCKS. TABLES THAT USE TABLE LOCKS
// CAN BE PHYSICALLY REDONE AND UNDONE. TABLES THAT ALLOW MULTIPLE WRITERS
// MUST ALSO BE ABLE TO USE LOGICAL UNDO.
//
        // This is to ensure that we never attempt a redo on a directory
        // block for a different readOwner. The redo logic would have to be
        // smarter to handle this.
        // It needs to redo up to the last committed change that is less than
        // readOwner.roLastLsn.
        //
        if ((entry.buf[BlockHeader.Flags1] & BlockHeader.DirectoryBlock) == 0 || entryLsn > roLastLsn || entry.owner
== writeOwner) {
            bestLsn = entryLsn;
            bestEntry = entry;
            bestDelta = delta;
            bestInRange = true;
            bestOwner = entry.owner;
        }
    }
}
// if (Diag.CHECK && bestEntry == null)

```

```

//      Diagnostic.println("---Passing up: "+block+" "+b1+" "+b2+" "+b3+" "+b4+" "+Long.toHexString(entryLsn)+"
"+Long.toHexString(roOldestLsn));
//      Diagnostic.check(bestEntry != null);
//      }
//      else {
//      if (Diag.CHECK && entry.block == block) {
//      Diagnostic.println("Passing up: "+block+" "+b1+" "+b2+" "+b3+" "+b4);
//      }
//      }
//      entry = entry.next;
//      }
// Its possible that the lsn of the writeOwner could not be safely
// read (see canCopy() above. If the writeOwnerEntry was found,
// and no other candidates are found, use the writeOwnerEntry.
// The copy operation below will retry until it can get the block
// contents including the lsn.
if (bestEntry == null && writeOwnerEntry != null) {
    bestEntry = writeOwnerEntry;
    bestOwner = writeOwner;
}
Diagnostic.check(bestEntry == null
    || bestEntry.owner == storeOwner.writeOwner
    || ((bestEntry.listenerFlags & ListenerFlags.READONLY_NOSHARE) == 0 &&
(bestEntry.listenerFlags & ListenerFlags.READONLY_COMPLETE) != 0));
// Before we leave the cache monitor, lets see if we can snag a writeOwner
// block if we need to.
//
if (bestEntry != null) {
    Diagnostic.check(bestEntry.cacheListener != null, "purged before");
    // Must be called after bestEntry.cacheFlags has CacheFlags.COPY flag
    // set otherwise cache.add() might purge the bestEntryBlock.
    //
    bestEntry.cacheFlags |= CacheFlags.NO_PURGE;
    retEntry = cache.add(block, writeOwner.dataStore.blockBytes, readOwner, saveListener);
    bestEntry.cacheFlags &= ~CacheFlags.NO_PURGE;
    Diagnostic.check(bestEntry.cacheListener != null, "null cacheListener");
    // This will keep other readOnlyTx connection threads from
    // messing with this block until we have completed any necessary
    // undo/redo. Must be cleared before returning from this method so
    // waiting threads can get access.
    //
    retEntry.listenerFlags |= ListenerFlags.NEEDLSN;
    retEntry.cacheFlags |= CacheFlags.NEEDCOPY;
    readOwner.copyFromOwner = bestEntry.owner;
    bestEntry.cacheFlags |= CacheFlags.COPY;
    writeCopy = true;
    CacheListener temp = bestEntry.cacheListener;
    if (cache.copyEntryIfAble(bestEntry, -1)) {
//      Diagnostic.println("++++ lucky dog copied: "+Integer.toHexString(bestEntry.listenerFlags)+"
"+Thread.currentThread().hashCode());

```

```
// Lucky dog, got it right away while in cache monitor!
//
bestLsn = Packer.unpackLong(retEntry.buf, BlockHeader.LOG_ID);
Diagnostic.check(bestEntry != writeOwnerEntry || (retEntry.listenerFlags |=
ListenerFlags.FROM_WRITEOWNER) != 0);
// Diagnostic.check((retEntry.listenerFlags |= ListenerFlags.FROM_WRITEOWNER) != 0);
if (Diag.CHECK && bestLsn == 0 && retEntry.block != 2 && retEntry.block != 3) {
    int listenerFlags = bestEntry.listenerFlags;
    int cacheFlags = bestEntry.cacheFlags;
    int accessGeneration = bestEntry.accessGeneration;
    int listenerAccessGeneration = ((SaveListener)bestEntry.cacheListener).accessGeneration;
    Diagnostic.println("listenerFlags: "+Integer.toHexString(listenerFlags)); //NORES
    Diagnostic.println("cacheFlags: "+Integer.toHexString(cacheFlags)+" old flags:
"+Integer.toHexString(cacheFlags)); //NORES
    Diagnostic.println("isWriteOwner: "+(bestEntry.owner==writeOwner)); //NORES
    Diagnostic.println("again: "+Long.toHexString(Packer.unpackLong(bestEntry.buf, BlockHeader.LOG_ID)));
//NORES
    Diagnostic.println(temp+" "+bestEntry.cacheListener+" accessGeneration: "+accessGeneration+"
"+listenerAccessGeneration); //NORES
    Diagnostic.println("immediate copy, bestLsn == 0 for block: "+block+" "+bestEntry.block); //NORES
    Diagnostic.check(BlockLog.dump(Diagnostic.out, "getReadBlock:", bestEntry.buf, 0, bestEntry.buf.length));
//NORES
    DataStore.sleep(2000);
    Diagnostic.println("again: "+Long.toHexString(Packer.unpackLong(bestEntry.buf, BlockHeader.LOG_ID)));
    DataStore.sleep(2000000);
    Diagnostic.check(bestLsn != 0, "immediate copy"); //NORES
}
if (Diag.CHECK && (retEntry.cacheFlags&CacheFlags.NEEDCOPY) != 0)
    cache.checkList(bestEntry);
Diagnostic.check((retEntry.cacheFlags&CacheFlags.NEEDCOPY) == 0);
}
}
}
if (Diag.READONLY) Diagnostic.println("BESTOWNER "+Thread.currentThread()+" "+(bestOwner ==
writeOwner)+" "+block);
if (retEntry != null) {
    yieldCount = 0;
    while (true) {
        synchronized(cache) {
            Diagnostic.check((bestEntry.listenerFlags&ListenerFlags.VIRGIN) == 0 || bestOwner == writeOwner, "virgin
violation");
            if ((retEntry.cacheFlags&CacheFlags.NEEDCOPY) == 0) {
// Diagnostic.println("++++ copied for me: "+Integer.toHexString(bestEntry.listenerFlags)+"
"+Thread.currentThread().hashCode());
                break;
            }
            else if (bestEntry.cacheListener == null) {
                Diagnostic.println("NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNnull cacheListener: "+FakeOwner.dump(bestEntry));
            }
            else if (bestEntry.owner != bestOwner) {
```

```

        Diagnostic.println("BBBBBBbestOwner: ");
    }
    else if (cache.copyEntryIfAble(bestEntry, -2)) {
//        Diagnostic.println("++++ copied: "+Integer.toHexString(bestEntry.listenerFlags)+"
"+Thread.currentThread().hashCode());
        break;
    }
    else {
//        Diagnostic.println("cant fix: "+bestEntry.block);
    }
}
//    if (Diag.READ_ONLY_TX) Diag.continueWriter();
if ((++yieldCount % 10) == 0) {
    try {
        Thread.sleep(10);
//        Diagnostic.println("waiting for block: "+block);
    }
    catch(java.lang.InterruptedException ex) {
        Diagnostic.printStackTrace();
    }
}
else
    Thread.yield();
//    Diagnostic.println("block: "+block);
}
Diagnostic.check((retEntry.cacheFlags&CacheFlags.NEEDCOPY) == 0);
bestLsn = Packer.unpackLong(retEntry.buf, BlockHeader.LOG_ID);
if (Diag.CHECK && bestLsn == 0 && retEntry.block != 2 && retEntry.block != 3) {
    Diagnostic.check(bestLsn != 0, "delayed copy, bestLsn == 0, block: "+block);
}
Diagnostic.check(bestEntry != writeOwnerEntry || (retEntry.listenerFlags != ListenerFlags.FROM_WRITEOWNER)
!= 0);
Diagnostic.check(!retEntry.owner.ownerCanPurge(retEntry), "ownerCanPurge Read Only Block");
}
else {
    if (Diag.READONLY) Diagnostic.println("ro fetchEntry "+block+" "+Thread.currentThread());
    retEntry = readOwner.fetchEntry(block, saveListener);
    Diagnostic.check((retEntry.cacheFlags&CacheFlags.NEEDCOPY) == 0, "NEEDCOPY on fetchEntry");
//    Diagnostic.check(Diag.hasMonitor(cache), "cache monitor");
    Diagnostic.check(!retEntry.owner.ownerCanPurge(retEntry), "ownerCanPurge Read Only Block");
    retEntry.listenerFlags |= ListenerFlags.NEEDLSN|ListenerFlags.READONLY_FETCHED;
    Diagnostic.check((retEntry.listenerFlags & ListenerFlags.NEEDLSN) != 0, "after fetchEntry NEEDLSN");
    bestLsn = Packer.unpackLong(retEntry.buf, BlockHeader.LOG_ID);
//    Diagnostic.println("fetchEntry lsn: "+Long.toHexString(bestLsn));
    highestLsn = bestLsn;
//    retEntry.roHighLsn = bestLsn;
    fetched = true;
}
boolean complete = false;
try {

```



```

// roHighLsn = retEntry.roHighLsn;
// Diagnostic.check(roHighLsn >= bestLsn);
Diagnostic.check(retEntry.owner != writeOwner);
Diagnostic.check(retEntry == null || (retEntry.cacheFlags&CacheFlags.NEEDCOPY) == 0);
if (bestLsn > highestLsn) {
    highestLsn = bestLsn;
}
if (bestLsn == highestLsn && bestLsn < roOldestLsn && bestOwner == writeOwner) {
    if (Diag.READONLY) Diagnostic.println("ro < startLsn "+block+" fromWriteOwner: "+writeOwner+" bestLsn:
"+Long.toHexString(bestLsn)+" startLsn: "+Long.toHexString(roOldestLsn)+" "+Thread.currentThread());
    if (Diag.RO) Diagnostic.println("RO_INRANGE_WRITER1 bn: "+block+" fromWriteOwner: "+writeOwner+"
bestLsn<startLsn: "+Long.toHexString(bestLsn)+"<"+Long.toHexString(roOldestLsn)+" "+Thread.currentThread());
    if (Diag.BASIC_RO) Diagnostic.println("inrange: "+block);
    complete = true;
    return retEntry;
}
if (bestLsn < roLastLsn) {
    if (bestLsn < roOldestLsn) {
        if (bestOwner == writeOwner || (fetched && writeOwnerEntry == null)) {
//      if (bestOwner == writeOwner || (fetched && !writeOwnerBlockInCache) || roHighLsn >= roOldestLsn) {
//      if (retEntry.roHighLsn < roOldestLsn) {
//      Diagnostic.check(bestOwner == writeOwner || fetched);
//      retEntry.roHighLsn = roOldestLsn;
//      }
            if (Diag.READONLY) Diagnostic.println("ro < roOldestLsn "+Thread.currentThread());
            if (Diag.RO) Diagnostic.println("RO_INRANGE_"+(fetched?"FETCHED":"WRITER2")+" bn: "+block+" bestLsn
< startLsn: "+Long.toHexString(bestLsn)+"<"+Long.toHexString(roOldestLsn)+" "+Thread.currentThread());
//Diagnostic.println("inrange2: "+block);
            complete = true;
            return retEntry;
        }
    }
    if (Diag.READONLY) Diagnostic.println("ro REDO < startLsn "+block+"
"+Integer.toHexString(retEntry.buf[BlockHeader.Flags1])+" bestLsn: "+Long.toHexString(bestLsn)+" startLsn:
"+Long.toHexString(roOldestLsn)+" endLsn: "+Long.toHexString(roLastLsn)+" "+Thread.currentThread());
    if (Diag.RO) Diagnostic.println("RO_REDO bn: "+block+" bestLsn < startLsn:
"+Long.toHexString(bestLsn)+"<"+Long.toHexString(roOldestLsn)+" endLsn: "+Thread.currentThread());
    if (Diag.BASIC_RO) Diagnostic.print("redo: "+block+" "+Long.toHexString(roLastLsn)+"
"+Long.toHexString(bestLsn)+" "+readOwner);
    redoBlock(retEntry, bestLsn, roUndoActives, roOldestLsn, roLastLsn);
    if (Diag.BASIC_RO) Diagnostic.println(" listenerFlags: "+Integer.toHexString(retEntry.listenerFlags));
}
else {
    if (bestLsn < roLastLsn) {
        goToLsn(bestLsn);
        next();
        // In this case the transaction terminated inbetween the start and end,
        // so we must have the correct block image.
        //
        if (roUndoActives.find(get(TxConst.CONID).getInt(), get(TxConst.SEQUENCE).getInt()) == null) {

```

```

        if (Diag.READONLY) Diagnostic.println(Long.toHexString(bestLsn)+" ro not in transaction "+block+" conid:
"+get(TxConst.CONID).getInt()+" sequence: "+get(TxConst.SEQUENCE).getInt()+" "+Thread.currentThread());
        if (Diag.RO) Diagnostic.println("RO_INBETWEEN bn: "+block+" startLsn <= bestLsn <= endLsn:
"+Long.toHexString(roOldestLsn)+"<="+Long.toHexString(bestLsn)+"<="+Long.toHexString(roLastLsn)+"
"+Thread.currentThread());
        if (Diag.BASIC_RO) Diagnostic.print("inbetween: "+block+" "+Long.toHexString(roLastLsn)+"
"+Long.toHexString(bestLsn)+" "+readOwner);
        if (Diag.BASIC_RO) Diagnostic.println(" listnerFlags: "+Integer.toHexString(retEntry.listenerFlags));
        complete = true;
        return retEntry;
    }
}

    if (Diag.READONLY) Diagnostic.println("ro UNDO "+" isWriteOwner: "+(bestOwner==writeOwner)+" "+block+"
startLsn: "+Long.toHexString(roOldestLsn)+" endLsn: "+Long.toHexString(roLastLsn)+" bestLsn:
"+Long.toHexString(bestLsn)+" "+Thread.currentThread());
    if (Diag.RO) Diagnostic.println("RO_UNDO bn: "+block+" isWriteOwner: "+(bestOwner==writeOwner)+"
startLsn/bestLsn/endLsn:
"+Long.toHexString(roOldestLsn)+"/" +Long.toHexString(bestLsn)+"/" +Long.toHexString(roLastLsn)+"
"+Thread.currentThread());
    Diagnostic.check((retEntry.listenerFlags & ListenerFlags.NEEDLSN) != 0, "before NEEDLSN");
    if (Diag.BASIC_RO) Diagnostic.print("undo: "+block+" "+Long.toHexString(roLastLsn)+"
"+Long.toHexString(bestLsn)+" "+readOwner);
    undoBlock(retEntry, bestLsn, roUndoActives, roOldestLsn, roLastLsn);
    if (Diag.BASIC_RO) Diagnostic.println(" listnerFlags: "+Integer.toHexString(retEntry.listenerFlags));
    Diagnostic.check((retEntry.listenerFlags & ListenerFlags.NEEDLSN) != 0, "NEEDLSN");
    Diagnostic.check((retEntry.cacheFlags&CacheFlags.NEEDCOPY) == 0, "undo NEEDCOPY");
}
complete = true;
return retEntry;
}

finally {
    retEntry.listenerFlags &= ~(ListenerFlags.NEEDLSN|ListenerFlags.VIRGIN);
    if (!complete) {
        synchronized(cache) {
            Diagnostic.println("Did not complete: "+FakeOwner.dump(retEntry));
            cache.flushEntry(retEntry.allocIndex, readOwner, retEntry);
        }
    }
    else
        retEntry.listenerFlags |= ListenerFlags.READONLY_COMPLETE;
}
}

```